CIS 500 — Software Foundations
Midterm II

Answer key

April 1, 2009

1. (5 points) Recall the definition of equivalence for `while` programs:

```
Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st':state), (c1 / st -→ st') ↔ (c2 / st -→ st').
```

Which of the following pairs of programs are equivalent? Write "yes" or "no" for each one. (Where it appears, `a` is an arbitrary `aexp` — i.e., you should write "yes" only if the two programs are equivalent for every `a`.)

(a)
```
X ::= A4
```
and
```
Y ::= A2 +++ A2;
X ::= Y
```
Answer: No

(b)
```
X ::= a;
Y ::= a
```
and
```
Y ::= a;
X ::= a
```
Answer: No

(c)
```
while BTrue do (X := !X +++ 1)
```
and
```
X := !X +++ 1
```
Answer: No

(d)
```
while BTrue do (X := !X +++ 1)
```
and
```
while BTrue do (X := !X --- 1)
```
Answer: Yes

(e)
```
while BFalse do (X := !X +++ 1)
```
and
```
skip
```
Answer: Yes

2. (5 points) Is this claim...

Claim: Suppose the command `c` is equivalent to `c;c`. Then, for any `b`, the command

```
while b do c
```

is equivalent to

```
testif b then c else skip.
```

... true or false? Briefly explain.

Answer: False. If `b` *evaluates to* `true` *and* `c` *does not change the value of* `b`*, then the first expression loops while the second may not (as long as c terminates).*

Grading scheme: 1 pt. for "false". 1 pt. for mentioning nontermination. 3 pts for correct counterexample (b evaluates to true and c does not modify b).

3. (5 points) Recall that a *program transformation* is a function from commands to commands. What does it mean to say that a program transformation is "sound"? (Answer either informally or with a Coq definition.)

   Answer:

   Informally: A program transformation is sound if `c` is equivalent to the result of transforming `c`, for every program `c`.

   Formally:

   ```
   Definition ctrans_sound (ctrans : com → com) : Prop :=
     forall (c : com),
       cequiv c (ctrans c).
   ```

   Grading scheme: -1 for variants on "every command produces the same result as its transformed version" (strictly speaking, they may also fail to produce a result). -2 to -5 for more serious mistakes or nonsense.

4. (7 points) Recall the definition of a valid Hoare triple:

   ```
   Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
   forall st st',
        c / st ⟶ st'
     → P st
     → Q st'.
   ```

   Indicate whether or not each of the following Hoare triples is valid by writing either "valid" or "invalid." Where it appears, `a` is an arbitrary `aexp`—i.e., you should write "valid" only if the triple is valid for every `a`.

   (a)
   ```
   {{True}} X ::= a {{X = a}}
   ```
   Answer: Invalid

   (b)
   ```
   {{X = 1}}
   testif (!X === a) then (while BTrue do Y ::= !X) else (Y ::= A0)
   {{Y = 0}}
   ```
   Answer: Valid

   (c)
   ```
   {{True}}
   Y ::= A0; Y ::= A1
   {{Y = 1}}
   ```
   Answer: Valid

   (d)
   ```
   {{False}}
   X ::= A3
   {{X = 0}}
   ```
   Answer: Valid

(e)
```
{{True}}
skip
{{False}}
```
Answer: Invalid

(f)
```
{{X = 5 ∧ Y = X}}
Z ::= 0; while BNot (!X === A0) do (Z ::= !Z +++ !Y; X ::= !X --- 1)
{{Z = 25}}
```
Answer: Valid

(g)
```
{{X = 1}}
while BNot (!X === A0) do X ::= !X +++ 1
{{X = 42}}
```
Answer: Valid

Grading scheme: 1 pt for each

5. (9 points) Give the weakest precondition for each of the following commands. (Please use the informal notation for assertions rather than Coq notation—i.e., write `X = 5`, not `fun st => st X = 5`.)

(a)
```
{{ ? }}  X ::= A5  {{ X = 5 }}
```
Answer: `True`

(b)
```
{{ ? }}  X ::= A0  {{ X = 5 }}
```
Answer: `False`

(c)
```
{{ ? }}  X ::= !X +++ !Y  {{ X = 5 }}
```
Answer: `X + Y = 5`

(d)
```
{{ ? }}  while A1 <<= !X do (X ::= !X---A1; Y ::= !Y---A1)  {{ Y = 5 }}
```
Answer: `Y - X = 5`

(e)
```
{{ ? }} while !X === A0 do Y ::= A1 {{ Y = 1 }}
```
Answer: `X=0 ∨ Y=1`

(f)
```
{{ ? }}
  testif !X === A0
    then Y ::= !Z
    else Y ::= !W
{{ Y = 5 }}
```
Answer: `(X=0 ∧ Z=5) ∨ (X<>0 ∧ W=5)`

Grading scheme: 1.5 points for each

6. (5 points) The notion of weakest precondition has a natural dual : given a precondition and a command, we can ask what is the *strongest postcondition* of the command with respect to the precondition. Formally, we can define it like this:

   `Q` is the strongest postcondition of `c` for `P` if:

   (a) `{{P}} c {{Q}}`, and

   (b) if `Q′` is an assertion such that `{{P}}c{{Q′}}`, then `Q st` implies `Q′ st`, for all states `st`.

   `Q` is the strongest (most difficult to satisfy) assertion that is guaranteed to hold after `c` if `P` holds before.

   For example, the strongest postcondition of the command `skip` with respect to the precondition `Y = 1` is `Y = 1`. Similarly, the postcondition in...

```
{{ Y = y }}
  if !Y === A0 then X ::= A0 else Y ::= !Y *** A2
{{ (Y = y = X = 0) ∨ (Y = 2*y ∧ y <> 0) }}
```

...is the strongest one.

Complete each of the following Hoare triples with the strongest postcondition for the given command and precondition.

(a) `{{ Y = 1 }}  X ::= !Y +++ A1  {{ ? }}`

Answer: `X = 2 ∧ Y = 1`

(b) `{{ True }}  X ::= A5  {{ ? }}`

Answer: `X = 5`

(c) `{{ True }}  skip  {{ ? }}`

Answer: `True`

(d) `{{ True }}  while BTrue do skip  {{ ? }}`

Answer: `False`

(e)
```
{{ X = x ∧ Y = y }}
  while BNot (!X === A0) do (
    Y ::= !Y +++ A2;
    X ::= !X --- A1
  )
{{ ? }}
```

Answer: `X = 0 ∧ Y = y + 2*x`

7. (12 points) The following program performs integer division:

```
div =
     Q ::= A0;
     R ::= ANum x;
     while (ANum y) <<= !R do (
        R ::= !R --- (ANum y);
        Q ::= !Q +++ A1
     )
```

If `x` and `y` are numbers, running this program will yield a state where `Q` is the quotient of `x` by `y` and `R` is the remainder. (We assume that program variables `Q` and `R` are defined.)

Fill in the blanks in the following to obtain a correct decorated version of the program:

```
                                   { 0<y } =>
                                   { 0=0 ∧ x=x ∧ 0<y }
Q ::= A0;
                                   { Q=0 ∧ x=x ∧ 0<y } =>
R ::= ANum x;
                                   { Q=0 ∧ R=x ∧ 0<y } =>

                                   { ___________________________________________ }

while (ANum y <<= !R) do (
                                   { ___________________________________________ } =>


                                   { ___________________________________________ }
   R ::= !R --- (ANum y);

                                   { ___________________________________________ }
   Q ::= !Q +++ A1

                                   { ___________________________________________ }
```

```
)
                                  { _________________________________________ } =>

                                  { x=Q*y+R ∧ R<y }
```

Answer:

```
                                  { 0<y } =>
                                  { 0=0 ∧ x=x ∧ 0<y }
Q ::= A0;
                                  { Q=0 ∧ x=x ∧ 0<y }
R ::= ANum x;
                                  { Q=0 ∧ R=x ∧ 0<y } =>
                                  { x=Q*y+R }
while (ANum y <<= !R) do (
                                  { x=Q*y+R ∧ y<=R } =>
                                  { x=(Q+1)*y+(R-y) }
   R ::= !R --- (ANum y);
                                  { x=(Q+1)*y+R }
   Q ::= !Q +++ A1
                                  { x=Q*y+R }
)
                                  { x=Q*y+R ∧ ~(y<=R) } =>
                                  { x=Q*y+R ∧ R<y }
```

Grading scheme: -1 for minor errors. -2 for each violation of the rules for forming decorated programs.

8. (4 points) Suppose we change the initial pre-condition in problem 7 from `0<y` to `True` (i.e., we allow `y` to be zero). Does the specification now make an incorrect claim — i.e., is the Hoare triple

   ```
   {{ True }}  div  {{ x=Q*y+R ∧ R<y }}
   ```

   invalid, or is it valid? Briefly explain your answer.

   Answer: The specification remains valid: if `y` *is* `0` *at the beginning, the program will never terminate and the required condition for validity will hold trivially.*

   Grading scheme: 2 pts for noting program does not terminate when y=0; 2 pts for stating that Hoare triple is valid for nonterminating program.

9. (6 points) Recall the syntax...

   ```
   Inductive com : Set :=
     ...
     | CWhile : bexp → com → com
   ```

   ...and operational semantics of the `while...do...` construct:

   ```
   Inductive ceval : state → com → state → Prop :=
     ...
     | CEWhileEnd : forall b1 st c1,
         beval st b1 = false →
         ceval st (CWhile b1 c1) st
     | CEWhileLoop : forall st st' st'' b1 c1,
         beval st b1 = true →
         ceval st c1 st' →
         ceval st' (CWhile b1 c1) st'' →
         ceval st (CWhile b1 c1) st''
   ```

Suppose we extend the syntax with one more constructor...

```
| CLoopWhile : com → bexp → com
```

...written `loop c while b`:

```
Notation "'loop c 'while' b" := (CLoopWhile c b).
```

The intended behavior of this construct is almost like that of `while...do...` except that the condition is checked at the *end* of the loop body instead of the beginning (so the body always executes at least once). For example,

```
X ::= A1;
loop
  X ::= !X +++ A1
while
  !X <<= A1
```

will leave `X` with the value `2`.

To define the operational semantics of `loop...while...` formally, we need to add two more rules to the `Inductive` declaration of `ceval`. Write these rules in the space below.

Answer:

```
| CELoopWhileEnd : forall b1 st st' c1,
    ceval st c1 st' →
    beval st' b1 = false →
    ceval st (CLoopWhile c1 b1) st'

| CELoopWhileLoop : forall st st' st'' b1 c1,
    ceval st c1 st' →
    beval st' b1 = true →
    ceval st' (CLoopWhile c1 b1) st'' →
    ceval st (CLoopWhile c1 b1) st''
```

Grading scheme: 2 points for rules of the right form with the right conclusion; 2 points for getting the true/false the right way around; 2 points for evaluating the bexp after the command ran instead of before.

10. (6 points) Having extended the language of commands with `loop...while...`, the next thing we want is a Hoare rule for reasoning about programs that use this construct. Recall the rule for `while...do...`:

$$\frac{\texttt{\{\{P} \land \texttt{b\}\}} \quad \texttt{c} \quad \texttt{\{\{P\}\}}}{\texttt{\{\{P\}\}} \quad \texttt{while b do c} \quad \texttt{\{\{P} \land \texttt{\textasciitilde b\}\}}}$$

Write an analogous rule for `loop...while...`.

Answer:

$$\frac{\texttt{\{\{P\}\}} \quad \texttt{c} \quad \texttt{\{\{P\}\}}}{\texttt{\{\{P\}\}} \quad \texttt{loop c while b} \quad \texttt{\{\{P} \land \texttt{\textasciitilde b\}\}}}$$

Grading scheme: -3 for minor errors. 0 for major/multiple errors.

11. (4 points) Recall (from the review session on Monday) the small-step variant of the operational semantics of IMP. The `astep` and `bstep` relations (not shown here) are small-step reduction relations for `aexp`s and `bexp`s. The small-step relation for commands is defined as follows:

```
Inductive cstep : state → com → com → state → Prop :=
  | CSAssStep : forall st i a a',
    astep st a a' →
    cstep st (CAss i a) (CAss i a') st
  | CSAss : forall st i n,
    cstep st (CAss i (ANum n)) CSkip (extend st i n)
  | CSSeqStep : forall st c1 c1' st' c2,
    cstep st c1 c1' st' →
    cstep st (CSeq c1 c2) (CSeq c1' c2) st'
  | CSSeqFinish : forall st c2,
    cstep st (CSeq CSkip c2) c2 st
  | CSIfTrue : forall st c1 c2,
    cstep st (CIf BTrue c1 c2) c1 st
  | CSIfFalse : forall st c1 c2,
    cstep st (CIf BFalse c1 c2) c2 st
  | CSIfStep : forall st b b' c1 c2,
    bstep st b b' →
    cstep st (CIf b c1 c2) (CIf b' c1 c2) st
  | CSWhile : forall st b c1,
    cstep st (CWhile b c1) (CIf b (CSeq c1 (CWhile b c1)) CSkip) st.
```

Suppose we extend the syntax of commands with `loop...while...`, as in the previous two problems. What needs to be added to the definition of `cstep`?

Answer 1:

```
  | CSLoop : forall st b c1,
    cstep st (CLoop c1 b) (CSeq c1 (CWhile b c1) st
```

Answer 2:

```
  | CSLoop : forall st b c1,
    cstep st (CLoop c1 b) (CSeq c1 (CIf b (CLoop b c1) CSkip)) st
```

Grading scheme: 1 point for basic syntax, 3 for rule logic.

12. (12 points) Recall the following definitions from `Smallstep.v`:

```
Inductive tm : Set :=
  | tm_const : nat → tm
  | tm_plus : tm → tm → tm.

Inductive value : tm → Prop :=
  v_const : forall n, value (tm_const n).

Inductive step : tm → tm → Prop :=
  | ES_PlusConstConst : forall n1 n2,
        step (tm_plus (tm_const n1) (tm_const n2))
             (tm_const (plus n1 n2))
  | ES_Plus1 : forall t1 t1' t2,
        (step t1 t1')
     → step (tm_plus t1 t2)
             (tm_plus t1' t2)
  | ES_Plus2 : forall v1 t2 t2',
        (value v1)
     → (step t2 t2')
     → step (tm_plus v1 t2)
             (tm_plus v1 t2').
```

In class, we discussed the Progress Theorem:

> *Theorem:* If `t` is a term, then either `t` is a value or else there exists some term `t'` such that `t` steps to `t'`.

Write a careful informal proof of this theorem.

Answer:

Proof: By induction on `t`.

- Suppose `t = tm_const n`, then it is a value by `v_const`.
- If `t = tm_plus t1 t2` for some `tm`s `t1` and `t2`, then by the IH `t1` and `t2` are either values or can take steps under `step`.
  - If `t1` and `t2` are both values, then `t` can take a step by `ES_PlusConstConst`.
  - If `t1` is a value and `t2` can take a step, then so can `t`, by rule `ES_Plus2`.
  - Otherwise, `t1` can take a step. In this case `t` steps as well, by rule `ES_Plus1`.

Grading scheme: 1 point for induction, 2 for the base case, 3 for stating the IH in the inductive case. 6 points for the case analysis, reasoning, and clarity of the inductive case.